The Case for Kairos: The Importance of Moment and Manner in Software Engineering Communication*

Shreya Kumar
Notre Dame University
shreya.kumar@nd.edu

Charles Wallace
Michigan Technological University
wallace@mtu.edu

Abstract

Students preparing for software engineering careers need to be proficient in the mechanics of communication and experienced in the basic communication genres common to the profession. We argue, however, that this is not enough: students must also be prepared for the inventive, in-the-moment nature of real project communication. Choosing the right moment and manner for inventive discourse is the essence of kairos*, a long-standing concept in the field of rhetoric. We find similarities between the concept of kairos and the role of communication in agile software development methods.*

We argue for the need to address kairos in software engineering education. We present an approach, based on the concept of cognitive apprenticeship, that we have used in a team software project course with successful results. Finally, we pose two important challenges: how to evaluate kairotic awareness across a student's academic career and beyond, and how to make software engineering instructors feel comfortable covering communication topics.

1. Introduction

Successful computing projects rely on effective communication. This commonplace sentiment is echoed by computing practitioners and academics alike, and emphasized in educational standards [1, 2]. But what are the appropriate communication competencies for a computing curriculum? To be sure, mastery of mechanics (e.g. grammar, oratory) is part of the answer; proficiency in common communication genres (e.g. bug reports, sprint planning meetings) is another.

But there is another quality of effective communication that is more resistant to easy categorization: the ability to communicate in the right way at the right time. This concept has long been recognized and studied in the field of rhetoric, under its classical name *kairos* [3]. Communicators who are sensitive to kairos think of their communication choices as situation-dependent and use their rhetorical tools in a way that fits current goals and constraints, rather than adhere strictly to fixed templates and routines. *Agile* software approaches [4], in particular, appeal strongly, though implicitly, to kairotic awareness.

In this paper we make the case for kairos — or what we term "agile communication" — as a topic highly relevant to software engineers and deserving of a place within software engineering education. After introducing the concept of kairos (§2), we illustrate it with a couple of vignettes (§3), then explore its relationship to agile software development approaches (§4), We discuss the interventions that engage our students in kairotic discourse (§5) and place it within a theoretical framework of Cognitive Apprenticeship (§6). We conclude with a reflection on our experiences to date and our plans for future work (§7).

2. Kairos: an Agile Perspective on Communication

Kairos and other concepts germane to software project communication come from the theories and practices — both classical and modern — of *rhetoric*. While the term "rhetoric" in common parlance typically refers to language that is vacuous, insincere, or even deceitful, the study of rhetoric is in essence the study of *strategic communication*. Successful communication requires a strategy informed by an awareness of audience, a broad knowledge of potential genres, and sensitivity to the effects of style. We see a clean fit between rhetoric and software development. The software developer, like the rhetorician, can rely on the arts of knowing how to inquire, what questions to ask, in particular situations to make appropriate communications for a variety of audiences.

At the heart of kairos is a qualitative notion of time — "the right moment", which exists in a duality with *chronos*, or "measurable time". Chronos presents a

*This work is supported by the National Science Foundation under grant DUE-1504860.

URI: https://hdl.handle.net/10125/60192
ISBN: 978-0-9981331-2-6

vision of time as an objective, constant dynamic force, in contrast to the episodic, context-sensitive nature of the "kairotic moment". While often associated with "timeliness", other contextual properties such as place and manner are typically attributed to kairos as well.

Communication within a genuine software development community is typically complex, nuanced and varied, and participants make a range of choices — either explicit or tacit — about the design of their communication. Today's technological workplaces are made up of teams that are "all edge", spontaneously forming and refactoring; in such a world, a rote approach to communication, relying on set timing and set genres, is unrealistic [5, 6]. For the unreflective practitioner, unconscious communication choices may have unintended and unfortunate consequences; for the kairotically aware practitioner, each communication act is carefully designed for interchange of ideas and unambiguous decision-making.

3. Kairos in Software Project Communication: Two Vignettes

We offer two examples of communication in software development. The first, from a student project, indicates a rhetorical maturity on the part of the student that leads to fruitful communication with her client. The second, from an open-source software project, shows that even experienced developers can fail to take the timing and manner of their communication into account.

Vignette 1. A team of three software engineering students is collaborating on a project with their client and technical expert, Professor Hank Taylor from the Mechanical Engineering department. An earlier team had met with the client several times and tried but did not succeed in producing the code needed. The current software development team has taken over from them and is attempting to complete the project. Several weeks into the term, the current team finds itself facing similar issues to the first team: they are behind schedule in presenting a requested analysis of the legacy code.

Denise, leader of the team, chooses her moment: a one-on-one meeting with Dr. Taylor. At an explicit, official level, she is fulfilling a requirement to check in with the client, but the value of this meeting for the team is deeper. They need to synchronize their current understanding of the code with Dr. Taylor, who has significant familiarity with it. Furthermore, the team needs to (re)establish trust with the client.

In order to get her head around the existing code, Denise has compiled a data dependency and control flow chart (Figure 1) on a large sheet of paper. This hand-drawn chart constitutes the team's

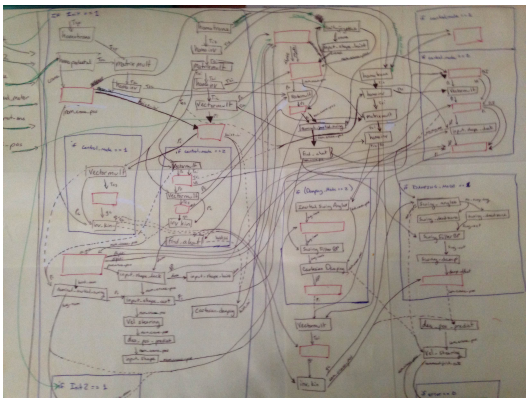

Figure 1. Denise's hand-drawn chart

understanding of the current code architecture. The visual representation is easy to update, but obviously rough and intimidating for the first-time reader. Denise brings the chart to the meeting and in fact makes it the focal point of the conversation with Dr. Taylor, using it to point at areas of the code, to articulate questions and responses. Together, they read the chart and mark it up as they go along.

Hank Taylor: Is this your chart?
Denise: Yes
HT: It looks exactly like his (another chart from a navy contractor)
D: No, his chart is much nicer.
HT: So have you folks started divvying it up?
D: This is where we need some help. So this is what happens in the code.
. . .
HT: Can you show me some example within the code? This is great. Don't throw this out. Is this hand-drawn?
D: Yes, I love the colors.

Denise's chart starts as a personal tool for developing her own understanding of the legacy code. In a deft rhetorical move, she then uses the chart at the crucial meeting with Dr. Taylor in an innovative way to facilitate a design and code specific technical discussion. This only works because Dr. Taylor is well versed in the code and has a mental model of the code, so the visual complexity of the chart does not intimidate him. Indeed, the complexity is a testament to the hard work of Denise and the team. Furthermore, Denise is present to mediate the discussion, and the sketch format allows for easy updates, so Dr. Taylor's input can be recorded.

Dr. Taylor is clearly enthused by Denise's presentation:
HT: Oh that is sweet! That makes sense now. So when this one is high, that value becomes high and this one

goes low, that value is low. I finally get it.
...
HT: I love it. I love it! The beauty of something like this is that I can understand it. Someone with a high level of knowledge of how the code or the function works can look at it and completely understand it.

His enthusiasm is evident in his call for a further communication moment:
HT: You know what it would take is about an hour and a half to sit down with the code and compare it with the blocks on the sheet. Let us set that up, and we don't all have to be there but the more the merrier.

The discussion mediated through the chart serves to disambiguate questions and explanations about code. The artifact also has a secondary effect, at what Austin famously termed the *perlocutionary* level [7]: expressing to the client the ability of the student to understand the code, giving her a chance to show her work and inspire Dr. Taylor's confidence in her abilities.

Ironically, when the same hand drawn chart was shown to students in a class activity, as a genuine artifact of software development, many students reacted negatively to it, calling it "messy and unprofessional" and "something [we] would never use with a client". In fact, this moment was a turning point: Dr. Taylor showed his approval of the chart and his respect for Denise's knowledge, exclaiming that Denise is now the person who knows the most about the code. From that point on, the project proceeded apace, and the relationship between client and students became strong.

This vignette offers a perfect teaching moment for future capstone project students: given the context and the client's reaction, they can understand the appropriateness of Denise's choices. The time it would take to produce a "professional" document would delay the conversation, and the kairotic moment would be lost.

Vignette 2. Audacity is a popular open-source, cross-platform recording and audio editing tool. The project is profoundly distributed in nature, with members from different countries using email as the primary means of communication and decision making. A core team of four to six people and receives code contributions from an assortment of programmers. Anyone in the developer list can start a discussion. Some of the participants on the forum may be regular contributors but not on the Core Team. The participation ranges from extremely regular to sporadic for some of the developers. The replies on the forum can be complex in style and are often deeply nested inline with varying degrees of quoting.

In the following email exchange, Benjamin, who has been regularly contributing for two years, has recently been inducted into the Core Team. Leland and Vaughn are long time contributors in the Core Team, and Vaughn is the most active on email. The project has just wrapped up a code freeze and the developers have some time to reflect on the bigger picture. Benjamin supports the idea of following standard code style:
The coding standard says: three spaces per indent
Why? Every other software project that I know uses either two, four, eight spaces per indent or tabs. Some code/text editors do not support three spaces per indent.

When Benjamin supports the question of enforcing style guidelines, he is "creating time" in a kairotic sense, choosing the period immediately after a code freeze as an opportunity for a coding process discussion.

An actively engaged and senior member of the open source community, Leland responds to Benjamin, sharing a link to a prior discussion in the community about the same topic:
Here's a little more background:
(weblink to prior conversation)
And I'm sure it was discussed way before that. For those that don't know Dominic was one of the original authors of Audacity.

Benjamin creates another moment: asking for this question to be considered again, as the constituents of that open source community had changed over time:
Maybe it's time to discuss it again. Old developers retired and new joined. So the overall preference could have been changed. Who of the Audacity developers has strong opinions regarding coding styles and who does not care as long it's consistent? Three spaces for indentation was a compromise between two and four. Who is for two spaces and who is for four?

From a kairotic perspective, Benjamin has chosen an inopportune moment in his second step, pushing the discussion forward before the more mature developers on the project have even given assent. As a challenge to the *status quo*, it calls for broad input — which is awkward in a medium like email. This raises the questions: what would be a good alternative to the poll taking, and how does one determine that the poll taking is done? In this project, email poll taking is a common practice. However, questions do not often get more than a few responses, and those responses can be a factor in the decision, dependent on who supports it.

After several emails, Vaughn responds to Leland:
V: We've had many developers over the years who participate for a short (sometimes long, often

infrequent) time, then move on, or become lurkers. (Ahem, like the guy who started this thread, and admitted he'd diverged from prior style when he contributed – much love!)
L: If this is a reference to me, then I was just doing what I thought you wanted...to keep quiet.
V: Thanks, Leland. I appreciate that, too.
Your original response was quite clear and nothing more needed to be said. Basically, the Audacity project is flexible, to some extent, when it comes to coding style. Why have you kept the thread alive? It's really simple...don't respond.
V: I kept it alive to engage the new posters, who apparently weren't getting what I said in my original response, and diverged into other topics.
Sorry it's not more pleasant, but I get frustrated repeating myself and being argued against the same thing. I'm glad you found it clear what I was saying, but yes, I did feed the troll.

As the discussion runs long and eventually becomes unpleasant, it no longer serves the originator in expressing his vision and attempt to foster a healthy discussion. Instead, some of the more experienced developers in the community express annoyance at the persistence of the discussion that they were trying to resolve quickly. In his choice, Benjamin has failed to take into account a history of related discussion, thereby alienating influential veterans of the project.

Both vignettes are examples of communication that have a significance beyond just the simple act of the communication, influenced by the context. Both are cases of communication which is strategically planned — planning a meeting, preparing a chart, starting a discussion — but where the participants also have to improvise aspects of the conversation, thinking more at a tactical level. The details of the communication require consideration, and there is no template or process that participants could follow to determine that. They can plan what the activity is going to be, then make smaller rhetorical choices in the moment of the activity.

4. Kairos in Software Engineering Practice and Education

Modern scholars have used kairos as a way to capture the essence of creativity in rhetoric — informally speaking, to break out of the confines of the *status quo* and craft communication that is valuable in the here and now. For instance, White argues for a place for kairos in contraposition to the deference to tradition characterized by *nomos*; he defines kairos as "the radical principle of occasionality establishing the living present as a point of departure for rhetorical invention" [8]. Applying this vision to a software development context, it is difficult to imagine a more accurate characterization of the philosophy behind agile development practices.

The principles of agile development [4] have resonance throughout the software industry. With this shift comes a change in how we approach communication. Agile developers are also agile communicators. At the heart of the agile approach is a recognition that requirements, priorities and obstacles in software projects are in constant flux. Consequently, agile methods encourage patterns of constant questioning, informing and debating. Agile developers must be unafraid to inquire about requirements, to critique design choices, and to provide reflective comments on the team's process.

While agile frameworks such as Scrum [9, 10] and Kanban [11] establish rituals and artifacts rooted in communication, these do not constitute a comprehensive, programmatic standard. Agile developers must be able to handle multimodal discourse (including written, oral and graphical communication through various media) and adapt to new communication situations, instead of relying on formal scripts and templates. In agile development, participants tailor the communication channels and genres they use dynamically to maximize value, rather than cleave to a predefined plan. Agile developers must be skilled rhetoricians, with a deep understanding of their communication options, and an ability to choose genre and style to suit the audience and purpose.

The stages of *invention*, *arrangement*, *style*, and *delivery* are particularly useful for teaching students to engage in active listening and critical analysis. Invention sets the requirements for the following three stages. It offers students a method for determining how to communicate most effectively with particular audiences in specific contexts, based on four sets of questions: audience, purpose, context, and perception. The arrangement information depends upon stakeholders' attitudes about the information and the students' purpose in conveying it. The style students choose to use — formal, informal, technical, colloquial — depends on both how they wish to be perceived, as well as their stakeholders' roles in the project. Finally, how students deliver the information — in an informal memo or more formal report — depends on the contexts in which users will apply the information.

To highlight the overlap and intersection of these stages, we use the metaphor of *communication cycles* [12] to describe the various documents that record and communicate the software development process. For instance, a typical cycle would include several technical

communication document genres that help to manage a project: an initial problem statement memo, followed by a project proposal, then a series of weekly progress reports that describe the successes and difficulties encountered as the project proceeds. Though some of these cycles can be codified into "best practices", there is always a time for rhetorical invention; for instance, Denise's hand-drawn chart began as a tool for individual learning, then as a worksheet enabling conversation with the client, then eventually became encoded as a formal document for future student developers.

Our work lies within a broad array of efforts to build instruction in communication directly into disciplinary courses, increasing the complexity and sophistication of the interventions systematically [13, 14, 15]. Falkner & Falkner define a methodology for integrating communication learning activities into computing courses [16]. Several authors stress the value of presenting students with authentic software communication genres, similar to what they will face as professionals [17, 18]. Hoffman *et al.* propose the use of workplace scenarios to approximate the complexity of real software project communication [19]. Furthermore, there is support in the computing education community for authentic team project experiences as a way to promote, among many other things, communication skills. A special issue of *ACM Transactions on Computing Education* on team projects feature several articles that discuss the benefits of exposure to complex communication contexts [20, 21, 22].

The project we describe here fits in this context of communication-intensive learning experiences grounded in the reality of professional software development. Our contribution, as we see it, is to step beyond the mastery of standard professional communication genres and investigate the inventive side of communication, as characterized by the concept of kairos. Our goal is to supply our students, the computing professionals of the future, with a rich set of rhetorical tools for the workplace *and* the know-how to use the right tool at the right time.

5. Addressing Communication in a Team Software Project Course

A decade of communication-related interventions of various kinds [23, 24, 25, 26] has led us to some conclusions about the proper timing and manner of such interventions — our own proper kairotic moment. First, there is a question of legitimacy: is this truly important to software engineers? For this reason, it is important to address issues of communication directly in disciplinary courses, rather than leaving them to auxiliary writing or communication courses, and to include problems on communication analysis as graded activities. Furthermore, any discussion of timely and appropriate communication must reside within a context of practice, and these contexts must be authentic: if possible, case studies pulled from real software project interactions. The idea of strategic communication as a topic in a software engineering classroom is already questionable to some students, and toy scenarios only exacerbate the situation. Even within an authentic communication context, there must be a framework within which an individual communication act can be analyzed and assessed. Otherwise, students may adopt a simplistic, negative attitude, too ready to make a blanket statement of "failure" while ignoring positive attributes.

Learning from several different week long interventions in different parts of the computing curriculum over a decade, the evolution of our approach was iterative by design. We went from covering entire student project case studies in a week to focusing on student selected moments, as we found that the entire case study story in a week and reading intensive interventions left some students feeling overwhelmed. The activities, which were initially fairly open ended to foster discussion, were then reformulated with structured, guided inquiry to avoid the pitfalls of only shallow, negative critique of all real, "messy" but possibly successful communication, allowing for more nuanced critique. Later, finding that just one week long intervention in a course did not allow students to absorb the material and adopt it in practicing their reflections, a scaffolding of different real world scenarios was created, including authentic industry scenarios as were requested by students. Realizing that students needed to practice these skills, different team reflection activities were incorporated over the course of many sprints. We discuss the course that was created as a culmination of practices accumulated from these lessons.

The target of our interventions is the third-year Team Software Project course, a precursor to the senior capstone project course [26]. We used our integrated course approach in two iterations of the course, where the demographics breakdown was 3 female students and 27 male students in one iteration and 5 female students and 34 male students in another iteration.

We start with case-based inquiries before student projects commenced and continued the conversation periodically through activities reflecting on their own teams. The ability to intervene regularly in the project course allows for a logical ramp-up from "textbook" communication practices to more complex case studies, and finally to the experiences of their own teams and those of their classmates. This set of practices offered in

this particular course appears to constitute our desired kairotic moment. The course allows them to work in teams and evaluate their teaming practices in a safe, sandboxed atmosphere before the higher stakes of capstone and industry projects.

In the Team Software Project course, students spend the first couple of weeks working on an introductory project to give students the preparation they need for their main semester long project: practice with the development tools they will be using, and instruction in the fundamentals of Scrum. In their main project, the students follow three Scrum sprints, each three weeks in length. Table 1 shows an outline of the course communication activities and other project milestones arranged chronologically, with the activities discussed in this paper in bold type. We share the quantitative and qualitative evaluation of our integrated approach in detail in our earlier work [26]. Here, we share some reflections on selected, kairotically relevant activities.

Table 1. Team Software Project course activities

Phase	Activity/Submission
Intro	Prototype + Use Cases + UML Diagrams
	Standard Scrum Practices
	Intro Project Demo + Deliverables
Prep	Main Project Proposal
	Sprint 1 Plan
	Student Team Project Communication
Sprint 1	**Open Source Project Communication**
	Requirement Elicitation - User Interviews
	Sprint Reflections (Individual)
	Sprint Reflections (Group)
	Sprint 1 Demo + Deliverables
	Sprint Peer Evaluations (Individual)
	Updated Project Scope
Sprint 2	Sprint 1 Retrospective + Sprint 2 Plan
	How "We" Scrum - How "They" Scrum
	Sprint Reflections (Individual)
	Sprint 2 Demo + Deliverables
	Sprint Peer Evaluations (Individual)
Sprint 3	Sprint 2 Retrospective + Sprint 3 Plan
	Daily Standup Assessment
	Sprint 3 Demo + Deliverables
	Sprint Peer Evaluations (Individual)
	Project Communication Report

5.1. Analyzing basic Scrum practices

We introduce communication analysis at the beginning of the second project through short guided inquiry activities. In the first session we ask students to analyze some standard Scrum communication practices (daily standup, burndown chart) that students have seen formal descriptions of and have had rudimentary experience with in the first, smaller project.

Students are given a short reading about these Scrum practices as homework, and in class they work together in groups to answer questions designed to allow students to examine different aspects of the assigned communication practice. The initial questions ask students to characterize the communication according to the traditional "Kipling questions" of Who, What, Where, When, How and Why.

Next, students are asked to conjecture how the communication act would be affected if one by one, different attributes of the communication act were changed. An example question from this activity is "Now imagine a scenario where the WHEN properties of a burndown chart were changed and it was updated twice in the project life cycle. How does that affect its use and relevance?"

The student groups then submit and share their answers with the rest of the class. Usually, a lively discussion follows where different groups discuss how they arrived at their answers. The purpose is to tease apart standard and prescribed communication acts and analyze their properties for their merits and demerits.

5.2. Analyzing student team communication

In the second guided inquiry activity, students are given excerpts of communication from real student software projects and asked to identify and analyze these communication acts. In one scenario — the subject of our first vignette — a student brings a rough hand-drawn control flow sketch (see figure 1) to a meeting with the client, then uses the sketch as a means for coming to understanding about the details of some legacy code. In another, a team leader sends an email message that reports results of a client meeting, includes sample code and delegates tasks to teammates. Both scenarios, in their imperfection, invite complex critique. For instance, the hand drawn chart can be seen as an "unprofessional" artifact to present to the client, but it can also be seen as an important catalyst for provoking detailed conversation between student and client. Similarly, the email message can be seen as effective in accomplishing a broad range of tasks, but packing so much disparate information into a single message can also be seen as unfocused and difficult for readers to parse.

This activity moves the class conversation from idealized descriptions of project communication to "messy" but more realistic artifacts, like the ones they will encounter and produce in their own projects. From experience, we know that students are primed to dismiss

the work of fellow students without serious analysis. To mitigate the risk of shallow, unmeasured criticism, we include written and oral cues to guide student inquiry in a more open-minded direction. For instance, in the case of the hand-drawn control flow chart, we frame the activity within a context where the student needs to demonstrate progress after some initial delay, check with the client expert, and quickly correct errors in her understanding as they arise in discussion. These cues shift student inquiry away from an assumption that only a flawless, "gold plated" diagram is acceptable in this circumstance. Explicitly acknowledging that informal, on-the-fly material is acceptable, particularly in situations of flux in requirements and assumptions, is critical at this moment.

5.3. Analyzing distributed collaborative project communication

In the third session, students examine email excerpts from another real case study: here, a geographically distributed open-source data visualization project where all communication happens over email. Students are asked to analyze an email exchange between the host of the group list and a programmer new to the group. Students are asked to identify (in general terms, but grounded in the given email material) the points of common understanding between the mentor and protégé and the points they are trying to resolve. Then the presence of an asymmetrical, mentor-protégé relationship is acknowledged, and students are then asked to identify features of the conversation that mitigate the risk of intimidation.

This activity exposes students to a truly authentic workplace scenario. This underscores the message that communication choices must be made even by seasoned professionals. This case study also introduces an interesting contextual constraint: relying solely on the asynchronous, textual medium of email. Students identified different communication patterns from the excerpts and were guided to identify underlying themes like implicit mentoring between participants.

5.4. How we Scrum – How they Scrum

Once the long-term projects in the course are underway, we ask students to reflect on their own practices. After the projects are underway, it is useful for each team to reflect on how they have implemented Scrum, and to critique the processes of other teams. This reinforces the idea that the software process and communication choices that a team makes are inevitably specific to that team, even within the confines of a particular process methodology.

This activity is intentionally conducted during the second sprint, when teams have been working together for over a month and have been through a whole sprint. At this point in the project, they have either deliberately developed or fallen into a set of communication practices. This activity helps them assess their own practices without the context of making a formal sprint plan and gives them exposure to how their peers have chosen to work together.

In "How we Scrum", team members work together to agree upon what communication practices they think define them by describing what their primary communication practices are and how or why they work for the team. Special attention is given to those related to division of work, communicating progress, conveying issues and soliciting help.

In "How they Scrum", teams interview members of other project teams to determine what communication practices the other teams engage in and how or why that works for them. Each team interview members from at least two other teams. The team members being interviewed use their answers from the How we Scrum activity. The students are encouraged to share their reflections with examples.

An excerpt from one "How We Scrum" response:
How we communicate: Primarily instant messaging, because we have very different schedules. When in person, we make good use of whiteboards to communicate ideas . . .
Issue resolution: Most of our issues are conceptual . . . so we tend to discuss the big issues at meetings (as they) are much easier to communicate in person . . .
Info/updates: We show each other new features or changes during every class period, and also after each standup meeting. We also have to be together to test the networking capabilities of the apps, which means that longer meetings are a regular occurrence. We use shared Google Docs to distribute diagrams and other files as we work.
Dividing up work: Each of us works on . . . a daily basis. Jack and Blake divide up work on the Android version based mostly on how much time they each have available to work on the project.
For conceptual issues, we try to have everyone in the discussion at the same time, so that we can all have input . . . When only a subset of the group is available . . . we make sure to bring it up for discussion again at the next meeting so that the others can have input.

This excerpt illustrates kairotic awareness among the students. They use a variety of media (instant messaging, Google Docs, whiteboards, paper sketches, face-to-face verbal communication), and the choice of

medium reflects the job at hand (instant messaging for ad hoc communication, Google Docs for more persistent information, in-person meetings to tackle conceptual issues). Also, when individual team members meet, the team is sensitive to the importance of getting the word out to the whole team.

5.5. Daily Standup Assessment

Towards the end of the third sprint, we ask students to assess their daily standup meetings [10]. Now they revisit something they were told to take as a given and something that is now meaningful as an activity to them, where they have developed rhythms for how much they share during the standup.

As part of this activity, each student anonymously rates every member of the team for the level of detail and the perceived value of the information offered by each team member in their update. Teams are then asked to reflect on the team's average graph (composed by combining the individual ones) why their average graph appears the way it does.

Some teams offered that their standup was not as valuable as other teams because all the team members worked together regularly and everyone knew what the current status of the project was, so the standup was practiced as a formality. Some teams indicated that the reason for their team's daily standup assessment being favorable — where they have a high average perceived value of the team's discussion during standup — was that the team often worked separately and standup meetings were times they synced with each other.

We observe that towards the end of the semester, teams reflect on simple practices using rich descriptions of context. Teams share more kairotically aware analyses. One student team reflects: *In some cases, it appears that the formal structure of the stand-up meetings works against us. It makes everyone feel like they need to have something to share, which leads to a lot of irrelevant information being shared if they have nothing more substantial . . . it would probably be best to use a somewhat less formal structure, so that people only share what they actually need to, rather than what they think is required. Overall though, the meetings continue to be a good way to keep everyone up to date with how the project is progressing."*

5.6. Evaluation

The course effectiveness was evaluated through quantitative and qualitative methods. A seven point Likert scale survey was used (response rate of 96.6%, confidence interval of 95%). The ten questions covered ease of performing the pattern and reflection activities, the importance of communication in industry and whether this course covered communication based instruction that is more relevant to them than other communication based courses.

For the survey statement, "What I learned about communication in this course was more relevant to my field than other courses about communication", the median response was 5.43 and the mode was 7(Strongly Agree), with a 0.56 margin of error. For the statement "This course has made me realize the importance of communication in the software industry," the median response was 5.5 with a mode response 7. Overall, we found that the students were able to learn how to do the communication activities easily and understood its relevance to practicum.

For a qualitative evaluation of the course, written group responses to homework were coded, on a rubric covering whether students discussed the impact of communication on their project, whether students critique their communication choices or discuss alternatives and whether students relate their communication experiences to prior ones. We noticed that even though all the teams were given the same instructions and project timelines, their practices and their reflections differed. Some teams followed Scrum practices formally, like doing Daily Standup meetings more than the required five a week, whereas some teams loosely followed Scrum. The team with the formal Scrum approach credited their success in large part to their strict adherence to process. We characterized depth of response as indicated by a greater number of coded reflections in the final project story assignment compared to Sprint plans earlier in the semester and in the quality of their written assignments. We found that the depth of reflection increased over the semester as the teams had more practice reflecting.

The specifics of our evaluation methodology and results are described in detail in our earlier work [26].

6. A Cognitive Apprenticeship approach

In this section, we discuss how we employ Cognitive Apprenticeship theory to drive our teaching of rhetorical skills to computing students [27, 28]. Cognitive Apprenticeship is a constructivist theory that attempts to capture the process by which apprentices gain skills from experts. Here we explain how our learning activities fit into a program of Cognitive Apprenticeship, and how we engage our students in these activities.

Modeling. Apprentices build a conceptual model of the task at hand by observation of experts at work. Our case studies provide insights into kairotically complex rhetorical situations, actions taken by the

participants, and their consequences. By selecting important moments in project communication, and by carefully articulating the problems and choices in these moments, or by guiding students toward identifying the problems and choices themselves, we as instructors can make the tacit explicit.

Scaffolding. Apprentices receive support in their early phases of learning, when they are still at a stage where they cannot complete the work themselves. Our instruction is carefully scaffolded so that students receive maximum support in early activities and gradually take on more challenging problems. We move from standard "textbook" rituals to more complex, grounded examples, and finally we ask students to reflect on and analyze their own practices once they have accrued sufficient experience.

Articulation. In an apprenticeship setting, learners are asked to verbalize or demonstrate tacit knowledge and thought processes — to "think aloud" in order to bring them into the open and define them. Through our communication pattern template, we reify communication acts, demonstrating that they have real structure and that the design choices have consequences [25]. By analyzing kairotic moments in case studies and in their own experiences, students are putting rhetorical knowledge into words, thereby forming a coherent conception of this knowledge.

Reflection. Apprentice learners take time to compare their own nascent problem-solving process with those of others. The "others" may include acknowledged experts, peers, and ultimately an internalized notion of expertise. Team Software Project students engage in reflection through the "How We Scrum" exercise, leading them to realize that they have indeed invented their own way of communication. The subsequent "How They Scrum" exercise challenges the principles behind their own chosen way and highlights the fact that communication is fundamentally inventive; there is more than one way.

Exploration. We use techniques from POGIL, which originated in undergraduate chemistry education [29] and has been introduced to computing disciplines through the CS-POGIL initiative [30, 31]. At the heart of POGIL is a *guided inquiry* learning cycle of *exploration*, *concept invention* and *application*. As an illustration of the cycle in practice, we refer back to our first communication exercise in Team Software Project (§5). We ask students to analyze standard Scrum communication practices, using our rubric as a guide to identify critical features of the communication strategies used (*exploration*). From these findings, students name patterns of communication and identify contextual characteristics that make the pattern suitable for application (*invention*). Next, students are asked to conjecture how the nature of a communication pattern would be affected if, one by one, different attributes of the communication act were changed (*application*).

7. Conclusion

Our interventions to date have demonstrated success in encouraging student to think analytically about the timeliness and manner of their communication [25, 26, 32]. There are limitations, however, to this work. First, evaluations to date have been restricted to our team software project; it is not clear to what degree student attentiveness to communication persists beyond the bounds of this course. Second, it remains to be seen how to equip software engineering instructors with the tools they need to teach topics in communication. We plan to address both of these issues in future work.

Students in our team software course are encountering a number of new practices and concepts in the span of a single semester — communication strategies being just one. While our interventions do seem to promote kairotic awareness within the span of the course, does the effect persist after the project is over? We plan to evaluate the work of students later in their senior capstone projects, using the qualitative techniques we have used in the team software course to identify moments where students design and reflect on their communication. We are also interested in the shift from student to professional: to what degree does an academic emphasis on communication help a young graduate in the workplace? We are planning a series of surveys of alumni to help answer that question.

Software engineering educators obviously have substantial technical skills and knowledge, and through experience in software development they also possess insights into how to communicate effectively. But without a framework or vocabulary to talk about communication, it is difficult to convey such insights to students. Moreover, the nature of human communication is inherently imprecise, yielding few hard and fast rules. It is easy to see why instructors may wish to leave discussion of communication to auxiliary courses taught by communication experts. Yet there is great value in covering communication directly in the software engineering class, since it provides authentic context and validates the importance of what might otherwise be a marginalized topic. Our use of patterns is an attempt to bridge the gap between the precision of computing and the multivocality of communication. To make our approach accessible to a wider audience, professional development material is a necessity, as well as a freely accessible and extensible library of

communication patterns.

References

[1] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2014.

[2] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser, "SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering," *Computer*, vol. 48, pp. 106–109, 2015.

[3] L. F. Bitzer, "The rhetorical situation," in *Rhetoric: Concepts, Definitions, Boundaries* (W. A. Covino, ed.), Allyn and Bacon, 1995.

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001.

[5] C. Spinuzzi, *Tracing Genres through Organizations: A Sociocultural Approach to Information Design*. MIT Press, 2003.

[6] C. Spinuzzi, *All Edge: Inside the New Workplace Networks*. University of Chicago Press, 2015.

[7] J. L. Austin, *How to Do Things with Words*. Oxford University Press, 1962.

[8] E. C. White, *Kaironomia: On the Will-To-Invent*. Cornell University Press, 1987.

[9] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Prentice Hall, 2002.

[10] K. Schwaber and J. Sutherland, "The scrum guide," *Scrum Alliance*, vol. 21, 2011.

[11] K. Leopold and S. Kaltenacker, *Kanban Change Leadership: Creating a Culture of Continuous Improvement*. Wiley, 2015.

[12] R. R. Johnson, *User-Centered Technology: A Rhetorical Theory for Computers and Other Mundane Artifacts*. SUNY Press, 1998.

[13] J. T. Havill and L. D. Ludwig, "Technically speaking: Fostering the communication skills of computer science and mathematics students," in *ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 185–189, 2007.

[14] A. Nylen and A. Pears, "Professional communication skills for engineering professionals," in *IEEE Frontiers in Education Conference*, 2013.

[15] K. Anewalt and J. Polack, "A curriculum model featuring oral communication instruction and practice," in *ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 33–37, 2017.

[16] K. Falkner and N. J. Falkner, "Integrating communication skills into the computer science curriculum," in *ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 379–384, 2012.

[17] C. D. Hundhausen, A. Agrawal, and P. Agarwal, "Talking about code: Integrating pedagogical code reviews into early computing courses," *ACM Transactions on Computing Education*, vol. 13, no. 3, 2013.

[18] J. E. Burge, G. Gannod, M. Carter, A. Howard, B. Schultz, M. Vouk, D. Wright, and P. Anderson, "Developing cs/se students' communication abilities through a program-wide framework," in *ACM Technical Symposium on Computer Science Education*, pp. 579–584, 2014.

[19] M. E. Hoffman, P. V. Anderson, and M. Gustafsson, "Workplace scenarios to integrate communication skills and content: a case study," in *ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 349–354, 2014.

[20] R. Pastel, M. Seigel, W. Zhang, and A. Mayer, "Team building in multidisciplinary client-sponsored project courses," *ACM Transactions on Computing Education*, vol. 15, no. 4, 2015.

[21] B. Bruegge, S. Krusche, and L. Alperowitz, "Software engineering project courses with industrial clients," *ACM Transactions on Computing Education*, vol. 15, no. 4, 2015.

[22] H. J. C. Ellis, G. W. Hislop, S. Jackson, and L. Postner, "Team project experiences in humanitarian free and open source software (hfoss)," *ACM Transactions on Computing Education (TOCE)*, vol. 15, no. 4, 2015.

[23] A. Brady, M. Seigel, T. Vosecky, and C. Wallace, "Addressing communication issues in software development through case studies," in *Conference on Software Engineering Education & Training (CSEET)*, 2007.

[24] A. Brady, M. Seigel, T. Vosecky, and C. Wallace, "Speaking of software: Case studies in software communication," in *Software Engineering: Effective Teaching and Learning Approaches and Practices* (S. D. H.J.C. Ellis and J. Naveda, eds.), IGI Global, 2008.

[25] S. Kumar and C. Wallace, "A tale of two projects: A pattern based comparison of communication strategies in student software development," in *Frontiers in Education (FIE)*, IEEE, 2013.

[26] S. Kumar and C. Wallace, "Instruction in software project communication through guided inquiry and reflection," in *Frontiers in Education (FIE)*, IEEE, 2014.

[27] A. Collins, J. Brown, and S. Newman, "Cognitive apprenticeship," *Thinking: The Journal of Philosophy for Children*, vol. 8, no. 1, pp. 2–10, 1988.

[28] A. Collins, "Cognitive apprenticeship," in *Cambridge Handbook of the Learning Sciences* (R. Sawyer, ed.), pp. 47–60, Cambridge University Press, 2006.

[29] T. Eberlein, J. Kampmeier, V. Minderhout, R. S. Moog, T. Platt, P. Varma-Nelson, and H. B. White, "Pedagogies of engagement in science," *Biochemistry and molecular biology education*, vol. 36, no. 4, pp. 262–273, 2008.

[30] C. Kussmaul, "Process oriented guided inquiry learning (pogil) for computer science," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pp. 373–378, ACM, 2012.

[31] H. H. Hu and T. D. Shepherd, "Using POGIL to help students learn to program," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 3, p. 13, 2013.

[32] S. Kumar, L. C. Ureel, and C. Wallace, "Agile communicators: Cognitive apprenticeship to prepare students for communication-intensive software development," in *AGILE'15*, 2015.